Query Execution

Two-Pass Algorithms

1. Two-pass algorithms.
   (a) Sort-based.
   (b) Hash-based.

Two-pass algorithms for relational algebra operations are applicable when participating relations do not fit in main memory. Two-pass algorithms are **not** *universal* (see constraints below), but are widely applicable. They also extend to multi-pass algorithms in a natural manner.

Sort-based Algorithms

All two-pass sort-based algorithms have the same basic structure:

Step 1. Segment-sort. Each relation participating in the operation is broken into segments of M pages each. Each segment is brought into main memory and sorted based on an *operation-dependent sort key*. Each segment is then flushed to disk.

This step involves **2 I/O operations** per each disk page occupied by the data files:

- 1 disk read: to bring the page into the buffer;
- 1 disk write: to flush the page to disk.

Step 2. Merge-sort. Each segment is allocated a slot in the buffer. All segments are scanned in parallel, and a merge-sort algorithm identifies on each step the tuple/tuples to be "consumed". The specifics of the relational algebra operation then determine the action performed on the tuple(s) and the output.

This step involves **1 I/O operation** per each disk page: a disk read to bring the page into the buffer.

For unary operations we have the following:

Cost:	$3 \cdot B(R)$
Constraint:	$B(R) \leq M^2$
Memory Use:	M

For binary operations, except for joins, we have the following:

Cost:	$3 \cdot (B(R) + B(S))$
Constraint:	$B(R) + B(S) \leq M^2$
Memory Use:	M

Specific algorithms are briefly discussed below.

Individual Operations

Duplicate Elimination. $\delta(R)$.

Step 1: Segment-Sort: The tuples are sorted on primary key R.

Step 2: Merge-Sort: During the Merge-Sort step the following actions are performed on each step:

1. The algorithm keeps track of the previous outputed tuple t_{prev}.
2. The tuple t with the smallest key value is selected and is removed from the page it resides on. If it was the last tuple on the page, next page (if one exists) from the same segment is read.
3. If $t \neq t_{prev}$, output t. (it will become t_{prev} on the next iteration. If $t = t_{prev}$ skip it.

Note: The algorithm leverages the fact that all merge-sort will process all copies of the same tuple in a row.

Grouping and Aggregation. $\gamma_L(R)$.

Step 1: Segment-Sort: The tuples are sorted on L.

Step 2: Merge-Sort: During the Merge-Sort step the following actions are performed on each step:

1. The algorithm keeps track of most recently seen values of attributes L in tuple l_{prev}. In addition, all aggregates requested by the operation for the group defined by l_{prev} are maintained in the tuple agg_{prev}.
2. The tuple t with the smallest key value is selected and is removed from the page it resides on. If it was the last tuple on the page, next page (if one exists) from the same segment is read.
3. Let $l = \pi_L(t)$.
4. If $l = l_{prev}$, update agg_{prev}.
5. If $l \neq l_{prev}$, output (l_{prev}, agg_{prev}). Set $l_{prev} := l$, and re-initialize agg_{prev}.

Set Union. $R \cup S$.

Step 1: Segment-Sort: The tuples in both relations are sorted on their primary keys.

Step 2: Merge-Sort: During the Merge-Sort step the following actions are performed on each step:

1. The algorithm keeps track of the previous outputed tuple t_{prev}.
2. The tuple t with the smallest key value is selected and is removed from the page it resides on. If it was the last tuple on the page, next page (if one exists) from the same segment is read.
3. If $t \neq t_{prev}$, output t. (it will become t_{prev} on the next iteration. If $t = t_{prev}$ skip it.

Note: Note that the set union algorithm looks **exactly the same** as the duplicate elimination algorithm. The only difference is that in the algorithm for union, we run merge-sort step on **both** R and S at the same time.

Intersection. $R \cap S$.

Step 1: Segment-Sort: The tuples in both relations are sorted on their primary keys.

Step 2: Merge-Sort: During the Merge-Sort step the following actions are performed on each step:

- For set intersection: for each next tuple turned up by the merge-sort procedure (i.e., tuple with the next smallest value of primary key), check to see if it appears in the other table. If yes, remove both from their respective pages, and output the tuple. If no, remove the tuple from the page, continue.
- For bag intersection: for each tuples turned up by the merge-sort procedure, count the number of its appearances in R and the number of its appearances in S. Output the tuple the number of times equal to the minimum of the two numbers.

Difference. $R - S$.

Step 1: Segment-Sort: The tuples in both relations are sorted on their primary keys.

Step 2: Merge-Sort: During the Merge-Sort step the following actions are performed on each step:

- For set difference: If the tuple with the smallest key is in S, remove it, continue. If the tuple with the smallest key is in R, check to see if it appears in S. If it does, remove it, continue. If it does not, output it, remove it, continue.

- For bag difference: Skip tuples with the smallest key turned up by merge-sort in S. For smallest key tuples in R, count the number of occurrences of the tuple in R, count the number of occurrences of the tuple in S, output the tuple the number of times equal to the difference between the two numbers, if it is positive. Remove all tuples, continue.

Sort-based Joins

Joins differ from set operation in one observation:

> The number of tuples from the two tables that share the join attributes may exceed available main memory.

Because of this, one has to be more careful with sort-based join algorithms.

Smiple Sort-based Join

We present an algorithm that assumes natural join: $R \bowtie S$. All other join variants can be computed in the same manner.

This version of the alrogithm presorts both R and S, and thus requires only 2 blocks in the buffer for the stage 2 scan. The remaining space can be used in processing.

Sort. Both R and S are sorted using a two-pass merge-sort algorithm. The sort is performed based on the join attributes.

Merge-Join. Merge-sort R and S. This involves using two buffer blocks to scan R and S. On each step, do the following.

1. Find the tuple with the current smallest sort key value k. If it *does not appear* in the other relation, remove it, continue.
2. If tuples with sort key value k are found in both relations, read into the buffer **all** tuples from R and S with sort key value k.
3. Perform memory-resident join on tuples with sort key k from R and S. Output all tuples resulting from this operation.
4. Clean buffer for next iteration.

Step 1, sort, costs $4 \cdot (B(R)+B(S))$ I/O operations: each disk page is read twice (once per merge-sort pass) and each disk page is flushed to disk twice (once per merge-sort pass). Step 2, merge-join costs $B(R) + B(S)$: each page is read once.

Cost:	$5 \cdot (B(R) + B(S))$
Constraint:	$B(R) \leq M^2$, $B(S) \leq M^2$, no join key value has more than $M - 2$ blocks of tuples in R and S combined
Memory Use:	M

The $B(R), B(S) \leq M^2$ are needed to perform merge-sort on step 1 in two passes. The remaining constraint assures that we can perform join for every key value in one pass.

More Efficient Join Algorithm (Sort-Join)

This algorithm is similar to the algorithms for other binary operations, is more efficient, but also – more constrained.

$R \bowtie S$.

Step 1: Segment-Sort: The tuples in both R and S are sorted on join attributes.

Step 2: Merge-Sort: During the Merge-Sort step the following actions are performed on each step:

1. Find the tuple t with the smallest value k of the sort key. Find all tuples in both R and S with the value k of the sort key.
2. Ouput the join of all tuples with the value k of the sort key, remove them from their respective disk pages, continue.

Cost:	$3 \cdot (B(R) + B(S))$
Constraint:	$B(R) + B(S) \leq M^2$, no join key value has more than M blocks of tuples in R and S combined
Memory Use:	M

If there are more tuples for a specific join key value, then nested-loops join can be used to produce the join result for this value of the key.